#6

РОССИЙСКОЕ АГЕНТСТВО ПО ПАТЕНТАМ И ТОВАРНЫМ ЗНАКАМ
(РОСПАТЕНТ)
ФЕДЕРАЛЬНЫЙ ИНСТИТУТ ПРОМЫШЛЕННОЙ СОБСТВЕННОСТИ

рег.No 20/12-722 "5 " октября 2000 г.



# СПРАВКА

Федеральный институт промышленной собственности Российского агентства по патентам и товарным знакам настоящим удостоверяет, что приложенные материалы являются точным воспроизведением первоначального заявления, описания, формулы, реферата и чертежей (если имеются) международной заявки № PCT/RU99/00340, поданной как в Получающее ведомство в соответствии с Договором о патентной кооперации в сентябре месяце 17 дня 1999 года (17.09.99).

Уполномоченный заверить копию
заявки на изобретение

Г.Ф.Востриков
Заведующий отделом

Копия для получающего ведомства

# PCT

## REQUEST

The undersigned requests that the present international application be processed according to the Patent Cooperation Treaty.

For receiving Office use only

PCT/RU 99 / 00340

International Application No.

17 сентября 1999 (17.09.99)

International Filing Date

RO/RU

МЕЖДУНАРОДНАЯ ЗАЯВКА PCT
PCT INTERNATIONAL APPLICATION

Name of receiving Office and "PCT International Application"

Applicant's or agent's file reference
*(if desired) (12 characters maximum)* **08883838WO**

**Box No. I TITLE OF INVENTION**

**System and Method for Producing a Verification System for Verifying Procedure Interfaces**

**Box No. II APPLICANT**

Name and address: *(Family name followed by given name; for a legal entity, full official designation The address must include postal code and name of country.)*

**NORTEL NETWORKS CORPORATION**
**World Trade Center of Montreal**
**380 St. Antoine Street West, 8th Floor**
**Montreal, Quebec H2Y 3Y4 Canada**

☐ This person is also inventor

Telephone No.

Facsimile No.

Teleprinter No.

State *(i.e. country)* of nationality: **CA**

State *(i.e. country)* of residence: **CA**

This person is applicant for the purposes of: ☐ all designated States ☒ all designated States except the United States of America ☐ the United States of America only ☐ the States indicated in the Supplemental Box

**Box No. III FURTHER APPLICANT(S) AND/OR (FURTHER) INVENTOR(S)**

Name and address *(Family name followed by given name; for a legal entity, full official designation. The address must include postal code and name of country.)*

**BURDUNOV, Igor**
**18 Abramtsevskaya St., Apt. 12, Moscow 127572, Russian Federation**

This person is:

☐ applicant only

☒ applicant and inventor

☐ inventor only *(If this check-box is marked, do not fill in below.)*

State *(i.e. country)* of nationality: **RU**

State *(i.e. country)* of residence: **RU**

This person is applicant for the purposes of: ☐ all designated States ☐ all designated States except the United States of America ☒ the United States of America only ☐ the States indicated in the Supplemental Box

☐ Further applicants and/or (further) inventors are indicated on a continuation sheet.

**Box No. IV AGENT OR COMMON REPRESENTATIVE; OR ADDRESS FOR CORRESPONDENCE**

The person identified below is hereby/has been appointed to act on behalf of the applicant(s) before the competent International Authorities as: representative ☒ agent ☐ common

Name and address: *(Family name followed by given name; for a legal entity, full official designation. The address must include postal code and name of country.)*

**KLIUKIN, Viacheslav Alexandrovich**
**Gowlings, International Inc.**

**B.Palashevsky 3, office 2, Moscow 103104, Russian Federation**

Telephone No.
**(7 502) 221 30 08**

Facsimile No.
**(7 502) 221 31 87**

Teleprinter No.

ПОЛУЧЕНО
17 СЕН 1999
ФИПС ОТД №20

☐ Mark this check-box where no agent or common representative is/has been appointed and the space above is used instead to indicate a special address to which correspondence should be sent.

PCT/RU 99 / 00340

**Continuation of Box No. III** **FURTHER APPLICANTS AND/OR (FURTHER) INVENTORS**

***If none of the following sub-boxes is used, this sheet is not to be included in the request.***

Name and address: *(Family name followed by given name; for a legal entity, full official designation. The address must include postal code and name of country.)*

**KOSSATCHEV, Alexander**
**154 Leninsky Pr., Apt. 54, Moscow 117571 Russian Federation**

This person is:
- [ ] applicant only
- [X] applicant and inventor
- [ ] inventor only *(If this check-box is marked, do not fill in below.)*

State *(i.e. country)* of nationality: **RU**

State *(i.e. country)* of residence: **RU**

This person is applicant for the purposes of
- [ ] all designated States
- [ ] all designated States except the United States of America
- [X] the United States of America only
- [ ] the States indicated in the Supplemental Box

Name and address: *(Family name followed by given name; for a legal entity, full official designation. The address must include postal code and name of country.)*

**PETRENKO, Alexander**
**56 Leningradsky Pr., Apt. 54, Moscow 117571 Russian Federation**

This person is:
- [ ] applicant only
- [X] applicant and inventor
- [ ] inventor only *(If this check-box is marked, do not fill in below.)*

State *(i.e. country)* of nationality: **RU**

State *(i.e. country)* of residence: **RU**

This person is applicant for the purposes of
- [ ] all designated States
- [ ] all designated States except the United States of America
- [X] the United States of America only
- [ ] the States indicated in the Supplemental Box

Name and address: *(Family name followed by given name; for a legal entity, full official designation. The address must include postal code and name of country.)*

**GALTER, Dmitri**
**17 Rue Des Alizes, Hull, PQ, J9A 3C3 Canada**

This person is:
- [ ] applicant only
- [X] applicant and inventor
- [ ] inventor only *(If this check-box is marked, do not fill in below.)*

State *(i.e. country)* of nationality: **CA**

State *(i.e. country)* of residence: **CA**

This person is applicant for the purposes of
- [ ] all designated States
- [ ] all designated States except the United States of America
- [X] the United States of America only
- [ ] the States indicated in the Supplemental Box

Name and address: *(Family name followed by given name; for a legal entity, full official designation. The address must include postal code and name of country.)*

This person is:
- [ ] applicant only
- [ ] applicant and inventor
- [ ] inventor only *(If this check-box is marked, do not fill in below.)*

State *(i.e. country)* of nationality:

State *(i.e. country)* of residence:

This person is applicant for the purposes of
- [ ] all designated States
- [ ] all designated States except the United States of America
- [ ] the United States of America only
- [ ] the States indicated in the Supplemental Box

- [ ] Further applicants and/or (further) inventors are indicated on another continuation sheet.

PCT/RU 99/00340

**Box No.V DESIGNATION OF STATES**

The following designations are hereby made under Rule 4.9(a) *(mark the applicable check-boxes; at least one must be marked)*:

**Regional Patent**

- [ ] **AP** **ARIPO Patent:** KE Kenya, LS Lasotho, MW Malawi, SD Sudan, SZ Swaziland, UG Uganda and any other State which is a Contracting State of the Harare Protocol and of the PCT
- [ ] **EA** **Eurasian Patent:** AZ Azerbaijan, BY Belarus, KZ Kazakstan, RU Russian Federation, TJ Tajikistan, TM Turkmenistan, and any other State which is a Contracting State of the Eurasian Patent Convention and of the PCT
- [X] **EP** **European Patent:** AT Austria, BE Belgium, CH and LI Switzerland and Liechtenstein, DE Germany, DK Denmark, ES Spain, FR France, GB United Kingdom, GR Greece, IE Ireland, IT Italy, LU Luxembourg, MC Monaco, NL Netherlands, PT Portugal, SE Sweden, and any other State which is a Contracting State of the European Patent Convention and of the PCT
- [ ] **OA** **OAPI Patent:** BF Burkina Faso, BJ Benin, CF Central African Republic, CG Congo, CI Côte d'Ivoire, CM Cameroon, GA Gabon, GN Guinea, ML Mali, MR Mauritania, NE Niger, SN Senegal, TD Chad, TG Togo, and any other State which is a member State of OAPI and a Contracting State of the FCT *(if other kind of protection or treatment desired, specify on dotted line)*.....

**National Patent** *(if other kind of protection or treatment desired, specify on dotted line)*:

- [ ] **AM** Armenia
- [ ] **AT** Austria
- [ ] **AU** Australia
- [ ] **BB** Barbados
- [ ] **BG** Bulgaria
- [ ] **BR** Brazil
- [ ] **BY** Belarus
- [X] **CA** Canada
- [ ] **CH und LI** Switzerland and Liechtenstein
- [ ] **CN** China
- [ ] **CZ** Czech Republic
- [ ] **DE** Germany
- [ ] **DK** Denmark
- [ ] **EE** Estonia
- [ ] **ES** Spain
- [ ] **FI** Finland
- [ ] **GB** United Kingdom
- [ ] **GE** Georgia
- [ ] **HU** Hungary
- [ ] **IS** Iceland
- [ ] **JP** Japan
- [ ] **KE** Kenya
- [ ] **KG** Kyrgyzstan
- [ ] **KP** Democratic People's Republic of Korea
- [ ] .....
- [ ] **KR** Republic of Korea
- [ ] **KZ** Kazakhstan
- [ ] **LK** Sri Lanka
- [ ] **LR** Liberia
- [ ] **LT** Lithuania
- [ ] **LU** Luxembourg
- [ ] **LV** Latvia
- [ ] **MD** Republic of Moldova
- [ ] **MG** Madagascar
- [ ] **MN** Mongolia
- [ ] **MW** Malawi
- [ ] **MX** Mexico
- [ ] **NO** Norway
- [ ] **NZ** New Zealand
- [ ] **PL** Poland
- [ ] **PT** Portugal
- [ ] **RO** Romania
- [X] **RU** Russian Federation
- [ ] **SD** Sudan
- [ ] **SE** Sweden
- [ ] **SG** Singapore
- [ ] **SI** Slovenia
- [ ] **SK** Slovakia
- [ ] **TJ** Tajikistan
- [ ] **TM** Turkmenistan
- [ ] **TT** Trinidad and Tobago
- [ ] **UA** Ukraine
- [ ] **UG** Uganda
- [X] **US** United States of America
- [ ] .....
- [ ] **UZ** Uzbekistan
- [ ] **VN** Viet Nam

Check-boxes reserved for designating States (for the purposes of a national patent) which have become party to the PCT after issuance of this sheet:

- [ ] .....
- [ ] .....
- [ ] .....
- [ ] .....

**In addition to the designations made above,** the applicant also makes under Rule 4.9(b) all designations which would be permitted under the PCT except the designation(s) of _____.

The applicant declares that those additional designations are subject to confirmation and that any designation which is not confirmed before the expiration of 15 months from the priority date is to be regarded as withdrawn by the applicant at the expiration of that time limit. *(Confirmation of a designation consists of the filing of a notice specifying that designation and the payment of the designation and confirmation fees. Confirmation must reach the receiving Office within the 15-month time limit.)*

PCT/RU 99 / 00340

**Box No. VI PRIORITY CLAIM** Further priority claims are indicated in the Supplemental Box ☐

The priority of the following earlier application(s) is hereby claimed:

| Country *(in which, or for which, the application was filed)* | Filing Date *(day/month/year)* | Application No. | Office of filing *(only for regional or international application)* |
|---|---|---|---|
| item (1) | | | |
| item (2) | | | |
| item (3) | | | |

*Mark the following check-box if the certified copy of the earlier application is to be issued by the Office which for the purposes of the present international application is the receiving Office (a fee may be required):*

☐ The receiving Office is hereby requested to prepare and transmit to the International Bureau a certified copy of the earlier application(s) identified above as item(s) : _____

**Box No. VII INTERNATIONAL SEARCHING AUTHORITY**

**Choice of International Searching Authority (ISA)** *(If two or more International Searching Authorities are competent to carry out the international search, indicate the Authority chosen; the two-letter code may be used):* ISA /**EPO**

**Earlier search** *Fill in where a search (international, international-type or other) by the International Searching Authority has already been carried out or requested and the Authority is now requested to base the international search to the extent possible, on the results of that earlier search. Identify such search or request either by reference to the relevant application (or the translation thereof) or by reference to the search request*

Country (or regional Office): Date *(day/month/year):* Number:

**Box No. VIII CHECK LIST**

This international application contains the following number of sheets:

| | | |
|---|---|---|
| 1. request | : | 4 sheets |
| 2. description | : | 36 sheets |
| 3. claims | : | 8 sheets |
| 4. abstract | : | 1 sheets |
| 5. drawings | : | 7 sheets |
| **Total** | : | 56 sheets |

This international application is accompanied by the item(s) marked below:

1. ☐ separate signed power of attorney
2. ☐ copy of general power of attorney
3. ☐ statement explaining lack of signature
4. ☐ priority document(s) *identified in Box No VI as item(s):*
5. ☒ fee calculation sheet
6. ☐ separate indications concerning deposited microorganisms
7. ☐ nucleotide and/or amino acid sequence listing (diskette)
8. ☐ other *(specify):*

Figure No. 1 of the drawings (if any) should accompany the abstract when it is published.

**Box No. IX SIGNATURE OF APPLICANT OR AGENT**

*Next to each signature, indicate the name of the person signing and the capacity in which the person signs (if such capacity is not obvious from reading the request)*

**Agent**
**V.A.Kliukin**

For receiving Office use only

| | | |
|---|---|---|
| 1. Date of actual receipt of the purported international application: I7 сентября I999 (I7.09.99) | | 2. Drawings: ☒ received ☐ not received: |
| 3. Corrected date of actual receipt due to later but timely received papers or drawings completing the purported international application: | | |
| 4. Date of timely receipt of the required corrections under PCT Article 11(2): | | |
| 5. International Searching Authority specified by the applicant: ISA/ EPO | 6. ☐ Transmittal of search copy delayed until search fee is paid | |

For International Bureau use only

Date of receipt of the record copy by the International Bureau:

*This sheet part of and does not count as a sheet of the international application.*

# PCT

## FEE CALCULATION SHEET

Annex to the Request

For receiving Office use only

PCT/RU 99 / 00340

International application No.

RO/RU

Date stamp of the receiving Office

Applicant's or agent's file reference 08883838WO

Applicant
Nortel Networks Corporation et al.

МЕЖДУНАРОДНАЯ ЗАЯВКА PCT
PCT INTERNATIONAL APPLICATION

### CALCULATION OF PRESCRIBED FEES

1. TRANSMITTAL FEE ..... 294 RUR [T] — 294 руб
2. SEARCH FEE ..... 2250 DEM [S] — 1899 DEM
   International search to be carried out by _____
   *(If two or more International Searching Authorities are competent in relation to the international application, indicate the name of the Authority which is chosen to carry out the international search .)*
3. INTERNATIONAL FEE

   **Basic Fee**
   The international application contains 56 sheets.
   first 30 sheets ..... 455 USD [$b_1$] — 455 USD
   26 (remaining sheets) x 10 (additional amount) = 260 USD [$b_2$] — 260 USD
   Add amounts entered at $b_1$ and $b_2$ and enter total at B ..... 715 USD [B] — 715 USD

   **Designation Fees**
   The international application contains 4 designations
   4 (number of designation fees payable *(maximun 11)*) x 105 (amount of designation fee) = 420 USD [D] — 420 USD
   Add amounts entered at B and D and enter total at I ..... 1135 USD [I] — 1135 USD
   *(Applicants from certain States are entitled to a reduction of 75% of the international fee. Where the applicant is (or all applicants are) so entitled, the total to be entered at I is 25% of the sum of the amounts entered at B and D.)*
4. FEE FOR PRIORITY DOCUMENT ..... 0 [P]
5. TOTAL FEES PAYABLE
   Add amounts entered at T, S, I and P, and enter total in the TOTAL box ..... 294 RUR, 2250 DEM, 1135 USD TOTAL — 294 руб / 1899 DEM / 1135 USD

☐ The designation fee is not paid at this time.

### MODE OF PAYMENT

☐ authorization to charge deposit account (see below) ☐ bank draft ☐ coupons
☐ cheque ☐ cash ☐ other *(specify)*:
☐ postal money order ☐ revenue stamps

### DEPOSIT ACCOUNT AUTHORIZATION *( this mode of payment may not be available at all receiving Offices )*

The RO/ _____

☐ is hereby authorized to charge the total fees indicated above to my deposit account.

☐ is hereby authorized to charge any deficiency or credit any overpayment in the total fees indicated above to my deposit account.

☐ is hereby authorized to charge the fee for preparation and transmittal of the priority document to the International Bureau of WIPO to my deposit account.

Deposit Account Number _____ Date *(day/month/year)* _____ Signature _____



*see Notes to the fee calculation sheet* ML 53612

# System and Method for Producing a Verification System for Verifying Procedure Interfaces

This invention relates to a system and method for producing a verification system for verifying procedure interfaces.

## BACKGROUND OF THE INVENTION

A software system contains a functionally closed set of procedures. In order to ensure correct implementation of the software system, it is desirable to determine a software contract, i.e., elements and functional specifications of external interfaces of the software system, and carry out conformance testing of the software contract implementation. Since the elements of the software contract are procedures, it is in fact Application Programming Interface (API) testing.

A kernel of an Operating System (OS) comprises API. For example, a Support Operating System (SOS) is a real-time OS for a Digital Multiplexing Switch (DMS) for a communication system. SOS comprises a plurality of processes for supporting the operation of DMS. The lowest layer of SOS is SOS kernel. The SOS kernel allocates resources to the processes running under SOS. The SOS kernel also provides communication among these processes. The SOS kernel also creates, controls and removes these processes.

SOS supports more than 25 million lines of code for applications and utilities. Thus, it is critical that user procedure interfaces of the SOS kernel be stable and reliable for correct performances of the DMS switch. The SOS kernel consists of over 1,700 procedures or over 230,000 lines of source code. Thus, it is very complicated and time consuming processes to generate a system for verifying such complex procedure interfaces. There existed no automatic or semi-automatic mechanisms to aid such generation of a verification system.

At the same time, SOS continuously evolves. Also, SOS is often ported to new hardware and software platforms. While more than 75% of the kernel procedures are machine-independent, the remainder of the kernel procedures are very machine dependent. The remainder describes particularity of memory, inter-processor communication and communication with peripheral devices. Accordingly, when SOS evolves or SOS is ported to a new platform, the SOS kernel and its procedure interfaces are also modified. Thus, the verification system for the procedure interfaces of the SOS kernel also needs to be modified. However, there existed no automatic or semi-automatic modifying mechanisms to aid such modifications.

There are some systems proposed for building a verification process. One of such systems is Interactive Tree and Tabular Combined Notation (TTCN) Editor and eXecutor (ITEX). ITEX is a test environment for communicating systems. It includes a TTCN and Abstract Syntax Notation.1 (ASN.1) analysis and design tool, a test simulator and support for generation of complete Executable Test Suites (ETS). In accordance with ITEX, a Test Suite is made up of Test Cases in form of tables. ITEX provides a set of highly integrated tools for development and maintenance of Abstract Test Suites (ATS) written in TTCN. ITEX supports phases of the test suite development including Test Case Generation, Editing, Verification, Validation and Execution. This toolset is integrated with the Specification Description Language (SDL) Design Tool (SDT), which is an environment for design of SDL specifications. Test suites described with TTCN can be transformed to the form that allows testing both implementation in some programming language and specification in SDL. However, this approach is unsuitable for API testing. TTCN does not permit declaration of pointers and other software entities that do not have textual (literal) representation. A major limitation of SDL-like specifications is their explicit form. This means that it is easy to build

models and prototypes based on them but it is very difficult to develop a system of constraints that define the union of all possible implementations.

Another example is the Algebraic Design Language (ADL)/ADL2. From formal specifications, ADL generates test oracles and skeletons for building test drivers and documentation. ADL uses not one of the popular specification languages but extensions of C and C++ languages. There are ideas on extensions of Java and other object-oriented languages aimed at developing software in "Design-by-Contract" fashion. However, despite the obvious advantages of better acceptance of such languages in the software engineering community, the concept, not to mention the common notation, is still far in the future. ADL has a limited range of API classes for which it can provides means for specifications and automatic test generation. ADL provides adequate tools for test generation automation only for procedures whose parameters allow independent enumeration and allows testing procedures one by one. This means that ADL omits procedures with dependent parameters, procedures that require testing in a group, e.g., "open-close", or those that require testing in parallel mode, e.g., "lock-unlock", or "send-receive".

Another example is formal derivation of Finite State Machines (FSM) for class testing proposed by L. Murray, D. Carrington, I. MacColl, J. McDonald and P. Strooper in "Formal Derivation of Finite State Machines for Class Testing", in Jonatthan P. Bowen, Andreas Fett, Michael G. Hinchey (eds.) ZUM'98: The Z Formal Specification Notation. 11-th International Conference of Z Users, Berlin, Germany, Sept. 1998, Proceeding, Lecture Notes in Computer Science, v. 1493, pp. 42-59. This work is at the research stage. The authors propose a scheme for organization of procedure group testing using Object-Z as specification language and C++ as programming language. The task of this work is stated to build test suites to verify conformance of the implementation

to the specification using formal specifications of the methods for a class. As a test coverage criterion, the union of two criteria is used: to cover all equivalency classes that represent the areas obtained as a result of partition analysis, and then, to check results on or near the boundaries. However, the authors of this work do not try to solve the problem of complete automation of test generation. Nor do they attempt to support any elements of the preparation phase with tools. Partition and boundary analysis is done manually according to the methodology proposed by the authors. In a similar way, they build the specification of oracles. Oracles, once compiled into C++, call target procedures and verify the conformance of the results to the specifications. This testing scheme is a framework that dynamically generates test sequences of procedure calls. The framework is controlled by the FSM description which represents an abstraction of a state transition graph of the test class. The authors describe the methodology of building specifications for the classes of states and transitions between them while considering the problem of exclusion of inaccessible states.

This approach needs the full description of the FSM that models the states of the system under test. The theoretical weakness of this approach is that it does not try to come up with a formal methodology to build transformation specifications. It is obvious that serious problems will be encountered when attempting to apply this approach to specifications of real-life complexity. In practical sense, it is clear that the process of test derivation from the specifications is mostly manual activity which limits its applicability to industrial software.

It is therefore desirable to provide a system and method which efficiently produces a verification system for verifying a procedure interface of different implementations.

SUMMARY OF THE INVENTION

The present invention uses formal specifications of a procedure interface, and generates test suites from the formal specifications. The formal specifications, and accordingly the test suites, are independent from implementation of the procedure interface. Thus, a verification system comprising the test suites may be efficiently produced for verifying the procedure interface of different implementations.

In accordance with an aspect of the present invention, there is provided a verification system generator for generating a verification system for verifying a Procedure Interface Under Test (PIUT). The verification system generator comprises means for producing formal specifications of procedures of PIUT in a form independent from implementation of the PIUT. The verification system generator also comprises a test source generator. The test source generator generates test sources based on the formal specifications. The generated formal specifications and test sources are stored in a repository.

In accordance with another aspect of the present invention, there is provided a method of producing a verification system for verifying a PIUT. First, formal specifications of the PIUT are generated in a form independent from implementation of the PIUT. Then, based on the formal specifications, test sources are generated. The test sources are used to generate a test suite in a form independent from implementation of PIUT for executing tests and analysing test results to verify the PIUT.

Other aspects and features of the present invention will be readily apparent to those skilled in the art from a review of the following detailed description of preferred embodiments in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be further understood from the following description with reference to the drawings in which:

Figure 1 is a diagram showing an embodiment of a verification system generator in accordance with the present invention;

Figure 2 is a flowchart showing an embodiment of a method for generating a verification system in accordance with the present invention;

Figure 3 is a flowchart showing steps of formal specification generation shown in Figure 2;

Figure 4 is a diagram showing a structure of a Support Operation System (SOS);

Figure 5 is a diagram showing an example of formal specification generation;

Figure 6 is a diagram showing an example of test suite generation;

Figure 7 is a diagram showing an example of test harness environment;

Figure 8 is a diagram showing an example of test execution features; and

Figure 9 is a diagram showing an example of test execution levels.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

There are different kinds of API entities, such as procedures, operations, functions, methods in C++ and subroutines in Fortran. In this specification, these terms are considered synonyms and all are called "procedure".

Referring to Figures 1 and 2, an embodiment of a verification system generator 1 and a method for generating a verification system 2 in accordance with the present invention are described. The verification system 2 is generated for verifying a procedure interface 4 of a System Under Test (SUT) 3.

The verification system generator 1 comprises means 12 for generating formal specifications, a test source generator 14 and a repository 16. As shown in Figure 2, the means 12 for generating formal specifications generates formal specifications of the procedure interface 4 (S10). Based on the formal specifications, the test source generator 14 generates test sources (S20). The generated formal specifications and the test sources are stored in the repository 16 (S30).

The test sources are used to generate a test suite 22. The test suite 22 is a set of programs and test data intended for the use in verifying the target procedure interface 4.

The formal specifications are generated in a form independent from implementation of the SUT 3. That is, the formal specifications do not depend on the implementation language, software or hardware of SUT 3, as further described later. The test sources that are generated based on the implementation independent formal specifications are also implementation independent. Accordingly, the test sources may be used on any implementation of the SUT or modified versions of the SUT.

The SUT 3 uses specific implementation language. The test sources are written in specification language that is independent from the implementation language. Accordingly, in order to execute the test sources on the SUT 3 to verify the procedure interface 4 of the SUT 3, the test sources are translated in language executable on the SUT 3 (S40). The translation is carried out by an implementation language compiler 18. The compiler 18 compiles some executable subsets of test sources in the specification language into programs in the implementation language of the SUT 3. The complied programs are specifications in implementation language that can be interpreted as description of some algorithms.

Thus, the generation of the verification system 2 is carried out in two stages. First, generation of

implementation independent programs is performed. Then, implementation independent programs are compiled into those in implementation language of the SUT 3. Such a two step generation method allows the means 12 for generating specifications and the test source generator 14 of the verification system generator 1 to be implementation-independent tools and, in particular, implementation-language-independent tools.

The complier 18 may be a part of the verification system generator 1 or may be provided separately from the verification system generator 1.

The compiled test sources form the test suite 22. As the test sources are implementation independent, the test suite 22 is also independent from the implementation of the target SUT 3, other than the language used. That is, the test suite 22 does not depend on the implementation software or hardware of SUT 3. By using the test suite 22, a test harness 20 including the test suite 22 and a test bed 24 is formed for verifying the procedure interface 4 of the SUT 3, as further described below.

The test suite 22 executes tests on the SUT 3 (S50) and analyses results of the tests to verify the procedure interface 4 (S60).

The verification system generator 1 "automates" test generation of real software for verifying a procedure interface 4 of an SUT 3. The expression "automation" used herein does not necessarily mean fully automated manipulation that creates ready for use test data, test sequences and other infrastructure for test execution and test result analysis. An "automated" process may include steps of manually writing some components in implementation language. When the total size of such manually developed components is small as a whole, the process may be considered "automated".

It is preferable that the test source generator 14 comprises a test driver generator 30 and a test case parameter generator 32.

The test case parameter generator 32 generates test case parameter sources for generating test case parameters. That is, the test case parameter generator 32 generates constant arrays and programs that generate and select needed test case parameters. The test case parameters are represented by these constant arrays and programs.

Based on the formal specifications, the test driver generator 30 generates test driver sources for generating test drivers. The test drivers execute tests on the SUT 3 using the test case parameters in implementation environments and analysing results of tests.

The test drivers comprise programs to execute and control testing of the procedure interface 4. The test case parameters are parameters of a test case. A test case is an instance of a tested procedure. A test case is defined by a procedure name and its parameters, i.e., test case parameters. Also, state of environment may be a factor of defining a test case. The test drivers use the test case parameters and execute test cases on the SUT 3 to verify the procedure interface 4.

The test driver generator 30 generates the test driver sources which, once compiled into the test drivers by the implementation language compiler 18, fulfil functions to initialize the procedure interface 4, prepare input values, call tested procedures with test case parameters, and receive test procedure results and analysis of the test results. In general case, the test driver sources are complex programs.

It is preferable that the test driver generator 30 generates the test driver sources that, once compiled into the test drivers, do not only pass some previously generated test case parameters to the SUT 3, but also control the state of the SUT 3. If the SUT state violates some conditions of the

test, the test drivers do not supply test parameters to the procedure interface 4.

As the formal specifications are implementation independent, the generated test driver sources and test case parameter sources are also implementation independent.

The test driver generator 30 preferably comprises a basic driver generator 34 and a script driver generator 36. The basic driver generator 34 analyses the formal specifications, and generates the basic driver sources comprising programs in implementation-independent language. The basic driver sources are used for generating a basic driver in implementation language. The basic driver is a test driver for a target procedure 4. The basic driver checks whether pre-conditions for the target procedure 4 hold for a given tuple of input parameters, calls the target procedure 4 with the given tuple of input parameter, records corresponding output parameters, and assigns a verdict on the correctness of the target procedure execution results. The basic driver preferably also collects information necessary to estimate test coverage or investigate reasons for a fault, as described below.

The script driver generator 36 generates script driver sources which describe sequences of calls to the basic driver with different test case parameters. The script driver sources are used for generating script drivers in implementation language. A script driver is a test driver for a target procedure or a set of target procedures. A script driver reads test options, generates sets of input parameters based on test options, and calls a basic driver with some set of input parameters. A script driver may also perform extra checking of the correctness of the target procedure execution results and assigns a verdict. A script driver may also check whether the test coverage is complete, and if not, it may continue to generate sets of input parameters and call the basic driver with this tuple.

The present invention may be suitably applied to generation of a verification system for arbitrary procedure interface of arbitrary systems. For example, the present invention is suitably applied to generate a verification system for procedure interfaces of a kernel of a Support Operating System (SOS) for a Digital Multiplexing Switch (DMS). The invention is hereinafter described mainly for verification of SOS kernel interfaces, but it is not limited to this application.

*Generating formal specifications*

The generation of the formal specifications of the procedure interfaces is further described referring to Figures 3 and 4.

The means 12 for generating specifications first provides a function (F12) for defining procedure interfaces of the SOS kernel (S12).

As shown in Figure 4, SOS 40 has SOS kernel 42 and SOS utilities 44. SOS 40 supports applications 46. SOS 40 is written using Nortel Networks Corporation's proprietary programming language called Protel, which is an example of the implementation, or target, language.

The SOS Kernel 42 comprises a plurality of procedures. The procedure interface defining function (F12) categorises the procedures of the SOS Kernel 42 into two groups: one group for those depending on implementation of SOS 40, and the other group for those independent from implementation of SOS 40. The procedure interface defining function (F12) then defines procedure interfaces to consist of procedures that are implementation independent. The defined procedure interfaces form a Kernel Interface Layer (KIL). KIL 43 does not depend on implementation and, in particular, on hardware special features of SOS 40. The procedure interfaces of KIL 43 are defined such that each procedure in KIL 43 performs one and only one service. No two procedures provide the same service.

Thus, KIL 43 comprises minimal and orthogonal procedures needed by upper layers of SOS 40 and applications 46. KIL 43 hides internal data structures and implementation details of the SOS kernel 42.

Based on the defined procedure interfaces of KIL 43, the means 12 for generating specifications provides a function (F14) for developing implementation independent description of the procedure interfaces of KIL 43 (S14).

The description developing function (F14) rigorously describes functionality of the procedure interfaces of KIL 43.

The implementation independent description may be developed using reverse engineering. The basic idea of the reverse engineering approach is a gradual "upwarding" of data representation in defined implementations. "Upwarding" is increasing the level of abstraction.

For example, as shown in Figure 5, it may be developed using source code 50 of the SOS kernel 42. The source code 50 is in the implementation language of SOS 40. The source code 50 is compiled into implementation independent language to generate a prime specification, i.e., implementation independent description 54. It is preferable to use an implementation independent language compiler 53 to carry out this compiling process automatically.

The implementation independent description may also be developed from documents or other information of the SOS Kernel 42.

As shown in Figure 3, the means 12 then provides a function (F16) for deriving formal specifications of KIL 43 from the implementation independent description (S16). In the example shown in Figure 5, the level of abstraction of the prime specification 54 is increased to generate a formal specification 56. This abstraction process 55 may be carried out manually.

It is preferable to use Rigorous Approach to Industrial Software Engineering (RAISE) to generate formal

specifications. RAISE Specification Language (RSL) is suitable to write formal specifications. RSL is supported by commercial tools for syntax and semantics checking, such as an EDEN-sintaxicaly oriented editor, a RAISE to ADA compiler, and a RAISE to C++ compiler.

Other RAISE features, e.g., axiom, algebraic specifications and channels may be used in semiformal considerations and explanations.

Also, it is preferable to use model-oriented specification in implicit form as the main form of specification. The implicit form describes a target procedure using pre-conditions and post-conditions of the target procedure.

The means 12 for generating specification may comprise a tool or a set of tools for providing above described functions for aiding a specifier to manually or semi-automatically generates the specifications. An example of such tools is the implementation independent language complier 53 as described above.

It is preferable to classify procedure interfaces of the target SUT by using the specifications. The following classification of procedures of a procedure interface is suitably used for generating a verification system for the procedure interface. The procedure interface classes include five main classes of procedures and some extensions of classes including procedures tested in parallel and expected exceptions. The classes are organized hierarchically. The first class establishes the strongest requirements. Each following class weakens the requirements. The requirements for the five classes are as follows:

KIND_1: The input is data that could be represented in literal (textual) form and can be produced without accounting for any interdependencies between the values of different test case parameters. Such procedures can be tested separately

because no other target procedure is needed to generate input test case parameters and analyse the outcome of the tests.

KIND_2: No interdependencies exist between the input items, i.e., values of input test case parameters. The input does not have to be in literal form. Such procedures can be tested separately. Examples of this class include procedures with pointer type input parameters.

KIND_3: Some interdependencies exist, however, separate testing is possible. Examples of this class include a procedure with two parameters in which the first one is array and the second one is a value in the array.

KIND_4: The procedures cannot be tested separately, because some input test case parameters can be produced only by calling another procedure from the group and/or some outcome of tests can be analysed only by calling other procedures. Examples of this class include a procedure that provides stack operations and that receives the stack as a parameter.

KIND_5: The procedures cannot be tested separately. Part of the input and output data is hidden and the user does not have direct access to data. Examples of this class include instances of Object-Oriented classes with internal states; and a group of procedures that share a variable not visible to the procedure user.

Exception raising extension of API classes: The specific kind of procedures raise exceptions as a correct reaction to certain input test case parameters. Examples of this class include a procedure that is supposed to raise an exception after dividing by zero. If zero received as an input parameter, then this procedure must not return any return code.

*Generating test sources*

The generation of the test sources is further described referring to Figure 6. Figure 6 shows an example of the test

generation for a KIL 43 using RAISE as implementation independent language.

The test source generator 100 comprises a basic driver generator 102, script driver generator 104 and test case parameter generator 106. In this example, the test source generator 100 uses UNIX, and the target SOS kernel 42 uses target language. The formal specifications 110 are generated in RSL. Accordingly, the test source generator 100 uses an RSL-target language complier 108 as an implementation language compiler.

The main source of the test source generation is the RAISE specifications 110. The RAISE specifications 110 are written in RSL. The RAISE specifications 110 may be those generated by the means 12 for generating specifications shown in Figure 1 or those stored in the repository 116.

The basic driver generator 102 receives the specifications 110. The basic driver generator 102 is a tool for generating basic driver sources, i.e., RSL basic drivers 103. The RSL basic drivers 103 are testing procedures in RSL. The basic driver generator 102 executes analysis of the RAISE specifications 110. Based on the analysis results, the basic driver generator 102 generates testing procedure programs comprising the RSL basic drivers 103. That is, the basic driver generator 102 generates, as the RSL basic drivers 103, programs for checking input test case parameters, calling tested procedures, tracing and analysing the test results, assigning a verdict of the outcome, and outputting trace information.

The basic driver generator 102 preferably also generates source for test case parameter generation 109. The source 109 for test case parameter generation preferably includes source for partition analysis, as described below.

The results 103, 109 of the basic driver generator 102 are fully completed in RSL sources. RSL generated sources do

not require any customization as they are implementation independent.

The RSL basic drivers 103 generated by the basic driver generator 102 are compiled by the RSL-target language compiler 108 into basic drivers 122 in the target language. The basic drivers 122 comprise target language procedures. Other than the language used, the RSL basic drivers 103 and the basic driver 122 in the target language are the same.

For each procedure in KIL 43, one basic driver 122 is generated. Each basic driver 122 provides direct call of a target procedure in KIL 43, and provides common facilities to test the target procedure. That is, each basic driver 122 takes input test case parameters for KIL 43, and checks pre-conditions of the target procedure. If the pre-conditions are correct, the basic driver 122 makes the call of the target procedure, and checks post-conditions of the target procedure.

The basic drivers 122 may carry out test result analysis by recording execution outcomes and comparing them with required outcomes. The basic drivers 122 may provide the result of the analysis as a verdict. The verdict may be either "passed" or "failed". The "passed" verdict means that no error is detected. The "failed" verdict means that an error is detected.

The basic drivers 122 may have a test oracle to automatically perform the analysis of the test outcome. The test oracle is a program that assigns a verdict on the correctness of outcome for the target procedure. The test oracle is similar to post-conditions. Both the test oracle and the post-conditions have Boolean functions. They have the same parameters, and return "True" if the target procedure produces a correct result and "False" otherwise. Accordingly, the test oracles can be generated once the post-conditions are generated.

The test result may depend on the SOS state and the history of SOS functioning. In order to fulfil its function,

each basic driver 122 preferably also generates programs to support a model of SOS state. The model is used to check acceptability of test case parameters in different contexts and to analyse correctness of test results.

The test case parameter generator 106 receives the source for test case parameter generation 109 from the basic driver generator 102. Then, the test case parameter generator 106 generates test case parameter sources, i.e., RSL test case parameters 107. The RSL test case parameters 107 may be constant arrays or programs. The test case parameter programs are also fully completed RSL sources.

The test case parameter generator 106 may also generate test case parameter sources from the specifications.

The RSL test case parameters 107 are compiled into test case parameters 126 by the RSL-target language compiler 108. The test case parameters 126 are input parameters for procedures under testing. Therefore, they are used for basic driver procedures. The test case parameters 126 may include only numeric and/or boolean input parameters. For example, a KIL of SOS includes about 140 procedures which need only such input parameters. These procedures are called KIND_1 procedures, as described above.

The script driver generator 104 receives the RAISE specifications 110, and generates script driver sources, i.e., RSL script drivers 105. The RSL script drivers 105 are compiled by the RSL-target language compiler 108 into script drivers 124 in the target language. Other than the language used, and the RSL script drivers 105 and the script drivers 124 in the target language are the same. The script drivers 124 are the upper level of the basic drivers 122.

Each RSL script driver 103 is a program for testing of a procedure or a group of procedures. It is a sequence of target procedures calls. The sequence may have serial or parallel composition. The sequence may have iterations. The RSL script drivers 103, once complied into the script drivers

124 by the complier 108, realize a scenario or script of testing.

The script driver generator 104 generates, as the RSL script drivers 105, programs to realize the sequences of procedure execution with different test case parameters. The script driver generator 104 generates the RSL script drivers 105 to have no direct interaction with target procedures. That is, the RSL script drivers 105, once complied into the script drivers 124, call the basic driver 122. One or more RSL script drivers 105 may be written to be called by procedures which function as suppliers of test case parameters 126, or procedures that allow a system operator to control a procedure group testing.

The script driver generator 104 may also generate programs to check the verdicts of the basic drivers 122. The script driver generator 104 may also generate programs to assign script driver own verdicts based on the basic driver verdicts.

It is preferable that the script driver generator 104 uses script driver skeletons 112 in addition to the specifications 110. The script driver skeletons 112 describe general scheme of script drivers. That is, each script driver skeleton contains an algorithm of a script driver. The script driver skeletons 112 are specific to each kind of procedure interface.

Each script driver consists of declarations and a body. The declarations include import of the procedure under test and its data structure definitions and/or import of all data and types used in the specifications. The declarations are generated automatically based on the list of procedures under test and their specifications 110. The body of a script driver begins with the script driver option parsing. The options, as parameters of the script driver as a whole, determine the depth of testing, e.g., the level of test

coverage criteria, and some specific data like interval of values, duration of testing.

In the example shown in Figure 6, in order to generate an RSL script driver 105, the script driver generator 104 uses one of the skeletons 112 and the RAISE specifications 110. Union of the specifications 110 and skeletons 112 forms formal description of test suite sources. This formal description may be considered as a test suite specification. The test suite specification allows the generator 100 to define test coverage requirements, schemes of script drivers, and algorithm for checking target procedure behaviours.

The script driver skeletons 112 for a new target SUT may be manually developed or received from the repository 116. Before testing starts, the verification system carries out some initialization. For example, before testing write/read procedures, the verification system opens a file. Such initializations are written manually. After initialization is finished, the main part of the script driver begins.

In addition to specifications 110 and skeletons 112, the script driver generator 104 may also use some supplement sources, such as some instances of test case parameters values.

The script driver generator 104 may also use procedures that convert values derived from the RAISE specifications 110 into value formats used by the current version of SOS kernel 42. Because the specifications 110 are implementation independent, correspondence between the specifications 110 and implementation data structures is separately described. Thus, it is preferable to use some means for associating abstract objects with implementation objects. Some target language procedures convert data from their representation in implementation to and from their representation in the test suite 120. Such target language procedures may be used as the associating means. The target language procedures use post-

conditions of the procedure under test. The target language procedures may be manually developed.

These additional sources including manually written skeletons may be called "manually developed components". The size of manually developed components is not large compared to the automatically generated components in the verification system generator 100.

For KIND_1 procedures, full automation of test generation is possible. All other kinds generally need some additional effort for writing manually developed components. The effort gradually grows from KIND_2 to KIND_5. The extensions require more effort than the corresponding kinds themselves. Complexity and effort for the development of manually developed components is usually caused by the complexity of the script driver generation and debugging. All script drivers for different classes of procedures have similar structure. The main distinction is the distribution between automatically generated components and manually developed documents. The KIND_1 script driver is generated fully automatically, KIND_2 script driver is generated almost automatically and so on.

The scheme of a script driver is further described in more detail using an example of a KIND_5 script driver.

The KIND_5 script driver realizes a general algorithm for traversing an abstract Finite State Machine (FSM). This algorithm passes all states and all possible transitions between the states. Each transition corresponds to an execution of a procedure under test.

The algorithm of a script driver is related to the specification and does not depend on the implementation details outside the specification. The script driver algorithm does not have direct descriptions of the abstract FSM. The verification system generator 100 avoids use of direct descriptions because direct specification of the FSM requires extra efforts to generate.

Instead of a direct specification of FSM, the verification system generator 100 uses indirect, virtual representation of FSM. Such representation includes a function-observer and a function-iterator. The function-observer calculates on the fly the current state in the abstract FSM. The function-iterator selects a next procedure from the target procedure group, and generates a tuple of the input parameter values for this procedure.

The KIND_5 script driver algorithm is described in more detail. For example, a case of testing a procedure group is considered. After passing several FSM states, i.e., some target procedures have been called, the next transition is being made. This elementary cycle of testing starts by calling a function-iterator that selects the next procedure from the target procedure group, and prepares a tuple of input test case parameter values for this target procedure. If the function-iterators have managed to generate a new and correct tuple without violation of pre-conditions, then the script driver calls a corresponding basic driver with the tuple as actual test case parameters.

When the basic driver returns a verdict, the control script driver checks the verdict assigned by the basic driver. If the verdict is "False", i.e., an error has been detected, the script driver produces corresponding trace data and finishes. If the verdict is "True", i.e., the elementary test case passed, the script driver calls the function-observer. The function-observer then calculates a current state, logs the state and transition, and continues to traverse FSM.

Thus, all possible states and test the procedures with all needed sets of input parameters may be obtained. FSM is used here as a guideline to pass through all states the needed number of times.

As described above, the script drivers are preferably composed following the requirements of the corresponding skeletons. In this embodiment, overall, the verification

system generator 100 uses five skeletons needed for serial testing of API KIND-1 through KIND_5 and one skeleton for parallel testing. Based on a corresponding skeleton and the list of target procedures and specifications, the verification system generator 100 generates a script driver template for each class. A KIND_1 template is a ready-to-use program. The templates for the other kinds include several nests with default initiators and iterators. If a test designer does not need to add or improve anything in the nests, the template can be compiled by the RSL-target language compiler 108 and executed as a script driver 124. This situation is typical for a KIND_2 procedure interface. For other kinds, a test designer usually adds some specific initiators and iterators as RSL supplement 115. The test designer defines FSM state observer for the script drivers of KIND_4 and KIND_5.

In the generator 100, all kinds of generation by generators 102, 104, 106 produce results 103, 105, 107, 109 in RSL. This means that "front end" of specification and verification technology is implemented in implementation language independent form. All generators 102, 104, 106 can produce the components 122, 124, 126 of the test suites 120 for systems implemented in arbitrary programming languages.

Compilation of generated sources 103, 105, 107 by the RSL-target language compiler 108 may be carried out when generation of all sources 103, 105, 107 is completed. The RSL-target language compiler 108 translates executable subsets of RSL language into programs in the target language. Thus, the RSL-target language compiler 108 restricts RSL. These restrictions are typical for all RSL language compilers. For example, the RSL-target language compiler 108 does not treat explicit definitions of constants if the user does not define the concrete constant value but only defines limitations that restrict constant field of values.

The RSL-target language compiler 108 is implementation-language dependent.

The result of the RSL-target language compiler 108 is generally a group of complete target-language sections. This is a part of the target language module that consists of a few sections. For obtaining a target language program which is ready to execute, some target language sections with interface descriptions may be produced. Interfaces or behaviour of some procedures from SOS are written once and do not need to be rewritten repeatedly. The target language sections with interface descriptions may be produced manually. These target language sections may be called target language supplement 114.

In order to correctly use different generation/compiling tools, it is preferable to know interdependencies between modules of specifications and between results of generation/compiling, i.e., the target language sections, and other target language modules/sections that were manually developed or had been produced by other tools. These interdependencies may be represented by a graph. The complexity of such a graph of interdependencies depends on the size of the procedure interface under test.

For example, currently KIL consists of over 560 procedures divided into over 30 subsystems. For each subsystem, there exists, at least, a basic driver module, and as a whole there exist about 200 script driver modules. For each RSL driver, at least one target language module is generated and stored. Besides, the target language modules consist of a few sections and each section is stored in a separate file. As a whole, KIL requires over 10,000 files. In order to facilitate use of test generation/compiling tools, it is preferable to provide a work "manage" utility, as described later.

The basic drivers 122 invoked by the script drivers 124 are generated fully automatically. The only manually developed components called from basic drivers 122 are data converters of the RSL-target language compiler 108. As

mentioned above, the converters transform the model data representation into the implementation representation and vice versa. A model representation is distinguished from the implementation one by the level of abstraction. For example, models may use "infinite" representation of integers, maps, relations, and other data structures suitable for specification. Sometimes model representation is very similar to the implementation one. In this case, such transformation is done by a standard translation algorithm of the specification language into the implementation language.

The verification system generator 100 is suitably used for generating a verification system for a continual evolving SUT. SOS may be evolved in accordance with its life cycle. During evolution cycle, requirements, interfaces or behaviour of some procedures from the SOS kernel, and implementation of SOS are repeatedly modified. For each new version of SOS, it is necessary to develop a new version of verification system. Therefore, it is beneficial to automate process of regeneration of the verification system.

Life cycle of test suites 120 generated by the verification system generator 100 replicates life cycle of the SOS Kernel 42. Usually, only a few interfaces or behaviour of some procedures from the SOS kernel are modified. The verification system generator 100 provides a possibility to re-specify modified interfaces or behaviour of some procedures from the SOS kernel and then re-generate test suites 120, and in doing so to provide re-use of old manually developed components. Thus, the verification system generator 100 can automate test suites regeneration. Therefore, existence of manually developed components does not decrease actual level of automation of the verification system generation.

To support automatic regeneration of test suites 120, the verification system generator 100 preferably stores in the repository 116 all manually developed components developed for generating the test suites 120 separately from automatically

generated components. The manually developed components supplement automatically generated components. Therefore, process of the test suites components manually development may be called "supplement". Thus, the verification system generator 100 may use two kind of sources for generating test sources: formal specifications and some supplement sources. As automatically generated and manually developed components of the verification system generator 100 are stored separately, no manual changes in automatically generated components are needed. Therefore, the verification system generator 100 can eliminate need of customizing automatically generated files for each regeneration of the test suites 120.

To estimate effort for generating verification system, a volume of modified interfaces or behaviour of some procedures from the SOS kernel is first estimated. When no interface is modified during SOS evolution, then no test (re)generation is needed. In that case, only realization, i.e., implementation, of SOS is modified. Therefore, previous specifications 110 and previous test suites 120 can be used for validation of the new KIL.

When some interfaces or behaviour of some procedures from the SOS kernel are modified or added during SOS evolution, then corresponding specifications 110 need to be modified. When interface data structures are modified, in addition to specifications 110, some conversion procedures in the target language also need to be (re)developed. Those target language conversion procedures may be manually developed. In any case, some reasons for test plan modification may arise. For example, these modifications may be caused by wishes to increase amount of tests, decrease time of testing, to check correlation of some features for parallel execution and so on. In those cases, some manual modification to manually developed components may be needed. When manual modifications are completed, a test designer can automatically generate new test

suites 120 for validation of the new SOS kernel by using the verification system generator 100.

In a simple case, it may suffice to modify the specifications 110 of types of pre-condition or post-condition of a target procedure. When new modification of procedure behaviour does not imply on behaviour of other procedure, the generator 100 needs only to regenerate a basic driver for verification of the modified procedure. In a complicated case, the generator 100 may need to regenerate totally new test suites including new basic drivers and script drivers. What volume of test suite modification is required depends on dependencies inside of the specifications 110 and between separate parts of the specifications 110 and test suites components 122-126 generated from these parts. Existing "manage" utility may be used which automates regeneration and recompiling of new test suites, as described later.

In order to port a test suite 120 generated by the verification system generator 100 from one implementation language platform to another, the data converters need to be rewritten and a new RSL to implementation language compiler needs to be provided. Also, a new run-time support system for the test suites with new test bed functions needs to be provided.

It is preferable that the verification system generator 100 also generates data for test coverage estimation and test plan design. The data is preferably kept in the repository 116.

Test coverage measures the completeness of testing. Sometimes, test coverage is presented as percentage of checked test situations. Sometimes, it is a list of test situations that have been or should be checked by the test suites. Test coverage requirements present all possible test situations that must be covered by test suites execution. If test suites 120 meet the requirements, then "exhaustive" or "100%" test coverage is gained.

There is a difference between test coverage estimation for source code and for specifications. In the case of source code, a test situation is associated with a statement, branch of path in control flow graph of a program. In the case of specifications, the specifications are represented as logical expressions, i.e., boolean expressions. Thus, test situations are associated with branches and disjuncts of boolean expressions. Therefore, by using test situations for specifications, it is possible to define test coverage requirements for arbitrary specifications. This allows uniform notation for description and accounting of the test situations, coverage requirements and obtained coverage.

The test suites are generated based on the specifications. The specifications are implementation independent. Thus, the test coverage is preferably measured by means of an implementation independent way. For this purpose, the verification system preferably uses test coverage criteria which are based on the specifications.

In complex SUTs, "all test situations" may not be covered. Accordingly, it is preferable to group similar test situations in classes. In this case, exhaustive coverage may represent coverage of all classes of test situations. Test situations and their classes may be identified and classified based on implementation source code or some external descriptions of the procedures under test. When a so-called "black box" approach is taken to test SUTs, test situations and their classes are identified and classified based on knowledge of descriptions of the procedures under test.

The test coverage criterion is a metric defined in terms of implementation or specification. The most well known test coverage criteria in terms of implementation are:

Class 1 - all statements are passed; and

Class 2 - all branches are passed.

In the case of using the specifications for test coverage criteria definition, the so-called domain testing approach is

preferably used. The whole input space is partitioned into areas by the basic driver generator. Each area corresponds to a class of equivalence.

The source for test case parameter generation 109 generated by the basic driver generator 102 preferably includes source for the partition analysis. The partition determines the choice of one of the test generation techniques applicable to a procedure interface or an interface of a procedure group. The source for partition analysis includes a list of test situation classes that represent test coverage requirements, and initial data for partition analysis. The source for partition analysis is used to generate test case parameters 126.

The partition may be derived from the specifications that describe requirements on input and properties of outcome for target procedures. Both the requirements and properties are represented in pre-conditions and post-conditions of formal specifications in implicit form. Accordingly, the test coverage estimation can be carried out based on the implicit specifications. In this example, the average percentage of the test coverage of the verification system generated by the generator 100 for SOS KIL is 70% to 100% of statements in the implementation.

Furthermore, there are two levels of the test coverage criteria. The first one is the coverage of all branches in post-conditions. The second one is the coverage of all disjuncts, i.e., elementary conjunctions, in the Full Disjunctive Normal Form (FDNF) representation of the post-condition while taking into account the pre-condition terms. The verification system generator 100 allows automatic partitioning in terms of specification branches and FDNF. It is preferable to calculate accessible FDNF disjuncts and remove the inaccessible FDNF disjuncts using pre-condition design.

Monitoring of obtained test coverage is preferably conducted on the fly by script drivers 124. Based on this data, the script drivers 124 may tune testing parameters and/or testing duration.

As described above, the verification system generation consists of a sequence of steps. For a step, needed tools are used where possible. Some tools generate auxiliary data, others convert and union their output and generate resulting target language code. To facilitate usage of these tools, it is preferable to use "manage" utility. The "manage" utility is a tool which works on a standard structure of UNIX directory where the tools and sources for verification system generations and generated test suites are stored.

The "manage" utility works like UNIX "make". It analyses an interdependencies graph which represents interdependencies of the test suite components, and searches "inconsistencies" in the graph. Each "inconsistency" requires to make some kind of generation and compiling. After the "manage" utility removes all "inconsistencies", all needed test suites components become ready to use. The "manage" utility uses a set of UNIX scripts and UNIX "make" files. The "manage" utility uses description of paths for all files used. It is preferable to define a "standard" directory structure to store sources and generated files for generating verification systems, and directories of the verification system generator.

*Test execution and testing result analysis*

Referring back to Figure 1, the verification system 2 is generated by compiling the test suite sources using the implementation language complier 18 as described above. The generated test suites are then loaded into the target SUT 3. During the test execution, the contents of the repository 16 are used and updated.

In order to execute the tests on the SUT 3, eg, the SOS kernel, it is preferable that the verification system 2 uses a

test harness 20. The test harness 20 is a software system used in the process of test execution and communicating with a target SUT 3.

Figure 7 shows an example of the test harness environment using the test suites 22. The test harness 20 comprises the test suites 22 and test bed 24. The test suites 22 comprises test drivers 60 including basic drivers and script drivers, and test case parameters 62.

The test bed 24 is a run-time support and some infrastructure features for test execution to enable functionality of the test drivers 60. The test bed 24 is a resident component of the verification system 2 located in SUT image, e.g., SOS image 48. During the test execution, the test harness 20 and SOS Kernel 42 form an instance of SOS image 48. This image 48 is a binary code structure that may be executed on a DMS switch.

In order to facilitate test execution, it is preferable that the verification system 2 uses a "Test Integrated Environment (TIE)" 26, as shown in Figure 1. The TIE 26 comprises a set of tools for executing test execution features and test analysis features. Some of the tools may be stored in repository, and others may be stored in separate files, e.g., UNIX files. The tools of the TIE 26 are executed under UNIX on a front-end computer or other OS. The test harness 20 with the SOS Kernel 42 is executed on the target hardware, e.g., DMS.

Figure 8 shows test execution features 130. The test execution features 130 include test plans design 132, test plans execution 134 and testing result analysis 136.

A test plan is a program that defines the order of script driver calls with the given test options. The test plan also checks the script driver call conditions and termination correctness.

Each test plan fulfils testing of a set of subsystems of SOS Kernel 42. A subsystem is a set of procedures. An

elementary component of a test plan is a test sequence. A test sequence is an instance of script driver execution with corresponding parameters. A test plan defines a serial or parallel order of test sequences. The purpose of the test plan is to check behaviour of subsystems under test in definite compositions. The test plan controls and manages test sequence execution. This control is executed by feedback which executes analysis of trace information. An algorithm of the test plan is called a test script.

Besides the set of basic drivers, script drivers and test case parameters, the test harness may also contain a test plan interpreter, a repository and tools for test plan execution and querying the results kept in the repository. The repository contains information about all test executions, code coverage metrics for different procedures with different test coverage criteria, and all situations when test drivers assigned a negative verdict.

Such test plan interpreter, repository and tools may be provided separately from the test harness.

The test scripts are preferably described in a script language. The script language is a lightweight programming language. In general, all possibilities provided by the script language may be realized by means of implementation language or other programming language. But, if those implementation languages are used, in order to compose ready-to-use script drivers, a test designer would have to implement usual programs and have to face usual troubles of linking/loading/debugging. However the script language allows to facilitate producing large variety of test sequence compositions for trial of large variety of test situations. Thus, it avoids wasting of time for compiling, linking and debugging usual for implementation language programs.

Source materials for the test plan design 132 are script driver informal description and test coverage requirements for the target subsystem(s).

The script driver description includes script drivers options description of resources needed for different options. Besides, the description may declare possibility or impossibility of sequential or parallel composition of this script driver with other script drivers.

As shown in Figure 9, the test execution 134 uses a hierarchical structure 140 comprising a user level 142, test plan level 144, script driver level 146 and a basic script level 148.

At the user level 142, test plans are activated by the user or by means of batch file execution in a test program. This level is intended for users of the verification system. The users execute test plans and receive the analysis results.

At the test plan level 144, a script driver is launched by a test plan. This level is intended for test plan designers who are also users of the verification system.

The introduction of the test plan level 144 and use of the script language specifically facilitate producing large variety of test sequence compositions for trial of large variety of test situations.

The introduction of the test plan level 144 also provides a possibility of feedback usage for script driver monitoring. The verification system supports two kinds of feedback information generated by the test drivers: verdicts and trace events shown by test coverage.

At the script driver level 146, the basic driver is called by a script driver. This level is intended for test designers who generate or write test suite components.

At the basic driver level 148, an interface procedure is called by the basic driver. Nobody has any access to this level. The basic driver is generated automatically, as described above. This level fulfils common functions of target procedure behaviour checking and tracing.

In the testing result analysis 136, verdict analysis is simple and is not informative. Verdict may be "PASSED" or

"FAILED". All script drivers can analyse verdicts obtained from its governed basic driver.

Test coverage is a measured for estimation of quality, i.e., fullness, of testing. Direct goal of introduction of such measure is to decide if a test suite is complete or not. The test coverage indirectly defines a criterion of test selection. For test coverage analysis, the TIE 26 uses description of all possible test situations that can arise during a procedure or subsystem testing. The TIE 26 compares test coverage requirements stored in repository 16 and trace information obtained from SOS Kernel 42 during testing.

The test coverage requirements for the target subsystem(s) use knowledge of a set of test situations. Test situations can be used for feedback. For example, a test plan may iteratively launch a script driver with different options until a certain test situation has been passed. Test coverage requirements are preferably generated automatically by the basic driver generator 34, as described above.

The verification system 2 may not provide any special tool for the test plan design 132. The test plans may be placed in UNIX files.

For the test plan execution 134, the TIE 26 may provide a navigating tool 27. The navigating tool 27 provides a graphical user interface to browse specifications and testware to run test and analyse test results. The navigating tool 27 is executed on a front-end computer. It communicates with the target hardware 6 or with the target hardware emulator. In order to invoke a test plan, the user of the verification system 2 may push a section in the navigating tool graphic interface of the TIE 26.

During test execution, the test drivers in cooperation with the TIE 26 collects trace information generated by the test drivers. The trace allows to estimate completeness of the test coverage.

The verification system 2 is intended for trial of non-stable (unreliable) SUT functioning. In particular, when a failure of the SUT 3 or the verification system 2 has raised during a test plan execution, it is preferable that the TIE 26 does not stop process of testing. In that case, the TIE 26 preferably stores the received trace information, then reloads the SUT 3, and continues to execute testing in accordance with list of test plans. Such scheme of testing may be called "incremental testing".

It is easy for many users to use a navigating tool which is an interactive tool for specifications, testware browsing, test execution and test result analysis. However, manual test execution means may also be suitably used.

Test execution and test result analysis features are preferably grouped in Test Integrated Environment (TIE) as described above. A user interface is provided in the TIE by the navigating tool. All other components of the verification system are hidden by the shell of the navigating tool.

The navigating tool is intended to analyse test execution results, to navigate through the test suite directory tree and to browse test plans, script drivers, procedures and their specifications.

The user interface of the navigating tool may use a Graphical User Interface (GUI). It may be Hyper Text Markup Language (HTML) document sets and Common Gate Gateway Interface (CGI) scripts.

The navigating tool may provide choices to, e.g., view subsystems, procedures, script driver kinds, script drivers, test plans and directories tree structure. The test plans link may be used to navigate through test plans available in the test system and to observe results of the test execution. When the user selects a test plan and choose a server on which those tests should be run, the navigating tool executes the tests using corresponding CGI scripts.

To build the entire set of all test situations, a CGI script finds its subsystems for each procedure in a trace log. Also, for all subsystems from set constructed in the previous step, it merges their test situations to build an available test situations set. The user may force the navigating tool to exclude subsystems selected by the user from consideration during report generation to decrease the test situations set. This is useful when test plan invokes script drivers that call the procedures, which are not under test. Such additional procedures prevent the user from true coverage percentage calculation.

The test coverage report may contain a number of and percentage of covered branches, a number of and percentage of covered disjuncts, and values about number of and percentage of covered procedures, branches and disjuncts.

The execution log may allow to, e.g., show time stamp, exclude script messages, exclude process died notification, exclude precondition failure, exclude incorrect post parameter message, hide basic driver fault, show no script driver warning, hide script driver fault, show no script driver message. The show time stamp option directs browser to precede each line in the report by time stamp value when given event was registered.

It is preferable that the verification system generator 1 also provides facilities for installation and maintenance. Installation consists of construction and filling a directory structure, setting environment variables and adding the user/administrator profiles. Utilities for installation are relatively simple.

The maintenance facilities covers updating of specifications and other sources for test suites generation; generation/compiling of test suites; and trial of new modified state of the verification system 2. The sources for test suites generation may be developed or modified remotely.

For each kind of above facilities, the verification system generator 1 may be provided with the corresponding utilities. If maintenance facilities do not have any restriction on resources, these utilities may be used in fully automatic manner. In general, modification of the verification system 2 may need to be performed step by step, for example, first generate a basic driver and second generate script drivers.

While particular embodiments of the present invention have been shown and described, changes and modifications may be made to such embodiments without departing from the true scope of the invention. For example, the present invention is described mainly using the verification system generator for verifying the SOS kernel. However, the present invention is suitably used for verifying a different system, such as a base level of call processing system, and a management system of tree-like store for queues with different disciplines. The present invention is mainly disclosed using RSL specifications. However, natural language documentation may also be used. Also, the present invention is mainly disclosed using UNIX. However, other operating systems may also be used.

**What is claimed is:**

1. A verification system generator for generating a verification system for verifying a Procedure Interface Under Test (PIUT), the verification system generator comprising:

means for producing formal specifications of procedures of PIUT in a form independent from implementation of the PIUT;

a test source generator for generating test sources based on the formal specifications; and

a repository for storing the generated formal specifications and test sources.

2. The verification system generator as claimed in claim 1, wherein the test source generator comprises:

a test driver generator for generating test driver sources which are used to produce test drivers that execute tests and analyse results of the tests in implementation environments of the PIUT; and

a test case parameter generator for generating test case parameter sources which are used to produce test case parameters used by the test drivers for test execution.

3. The verification system generator as claimed in claim 1, wherein the test driver generator includes:

a basic driver generator for generating basic driver sources, the basic driver generator analysing the formal specifications, and generating program sources of a basic driver for each PIUT; and

a script driver generator for generating a script driver source which describes a sequence of calls to the basic driver with different test case parameters.

4. The verification system generator as claimed in claim 3, wherein the basic driver sources are used to produce basic drivers in a form independent from implementation of the

PIUT, each of the basic drivers corresponding to each PIUT and, during the test execution, provides direct call of its corresponding PIUT, tests the corresponding PIUT, and traces and analyses test results of the corresponding PIUT.

5. The verification system generator as claimed in claim 3, wherein the basic driver generator further generates a list of test situation classes that represent test coverage requirements.

6. The verification system generator as claimed in claim 3, wherein the basic driver generator further generates initial data for partition analysis of the PIUT for the test case parameter generator to generate the test case parameter sources.

7. The verification system generator as claimed in claim 6, wherein the test case parameter generator receives the initial data from the basic driver generator, and generates programs for obtaining the test case parameters.

8. The verification system generator as claimed in claim 3, wherein the script driver sources are used to produce a script driver in a form independent from implementation of the PIUT, the script driver controls a sequence of calls to the basic drivers and the test case parameters during the test execution.

9. The verification system generator as claimed in claim 8, wherein the script driver further controls serial and parallel iterations of the test execution of the corresponding PIUT with different test case parameters.

10. The verification system generator as claimed in claim 3, wherein the script driver generator uses script driver skeleton description and PIUT prototype specification.

11. The verification system generator as claimed in claim 1 further comprising an implementation language compiler for compiling the test sources into an implementation language of the PIUT.

12. The verification system generator as claimed in claim 11, wherein the test source generator further comprises a manage utility which analyses interdependencies between the formal specifications and between the tests generated and compiled by the verification system and programs generated by other systems, searches inconsistencies, and removes the searched inconsistencies.

13. A method of producing a verification system for verifying a Procedure Interface Under Test (PIUT), the method comprising the steps of:

generating formal specifications of the PIUT in a form independent from implementation of the PIUT; and

generating test sources based on the formal specifications, which test sources are used to generate a test suite in a form independent from implementation of PIUT for executing tests and analysing test results to verify the PIUT.

14. The method as claimed in claim 13, wherein the step of generating formal specifications comprises the steps of:

defining PIUT;

developing implementation independent description of the PIUT; and

deriving the formal specifications based on the implementation independent description.

15. The method as claimed in claim 13, wherein the step of generating formal specifications uses Rigorous Approach to Industrial Software Engineering (RAISE) Specification Language (RSL).

16. The method as claimed in claim 13, wherein the step of generating test sources comprises the step of generating constant arrays and programs for generating and selecting needed test case parameters.

17. The method as claimed in claim 13, wherein the step of generating test sources comprises the steps of:

generating test driver sources which are used to produce test drivers that execute tests and analyse results of the tests in implementation environments of the PIUT;

generating test case parameter sources which are used to produce test case parameters used by the test drivers for test execution; and

creating test plans for carrying out tests based on the formal specifications.

18. The method as claimed in claim 17, wherein the step of generating test driver sources comprises the step of generating basic driver sources by analysing the formal specifications, and generating program sources of a basic driver for each PIUT.

19. The method as claimed in claim 18, wherein the step of generating test driver sources further comprises the step of generating script driver sources by creating skeletons describing general scheme of the script driver, and

generating script driver sources based on the formal specifications and the skeletons.

20. The method as claimed in claim 18, wherein the step of creating test plans comprises the step of describing scripts for test execution, the scripts defining test sequences, each test sequence being an instance of a basic driver with definite test case parameters.

21. The method as claimed in claim 20, wherein the step of describing scripts uses a special script language which facilitates serial and parallel combination of the test sequences.

22. The method as claimed in claim 17, wherein the step of creating test plans includes the step of generating data for test coverage estimation.

23. The method as claimed in claim 13 further comprising the step of storing the generated formal specifications and test sources in a repository.

24. The method as claimed in claim 23, wherein the step of storing comprising the steps of:

separating manually developed components from automatically generated components; and

separately storing the manually developed components from the automatically generated components.

25. The method as claimed in claim 24, wherein the step of generating test sources includes the step of re-generating test sources using the manually developed components when the PIUT is modified.

26. The method as claimed in claim 25, wherein the step of re-generating test sources is carried out automatically.

27. The method as claimed in claim 13 further comprising the step of compiling the test sources into an implementation language used by the PIUT.

28. The method as claimed in claim 27, wherein the step of compiling the test sources includes the step of describing correspondence between the formal specifications and implementation data structures of the PIUT.

29. The method as claimed in claim 27, wherein the step of compiling the test sources using converting procedures that convert values derived from the formal specifications into value formats used by the PIUT.

30. A method of verifying a Procedure Interface Under Test (PIUT), the method comprising the steps of:
generating formal specifications of the PIUT in a form independent from implementation of the PIUT;
generating test sources based on the formal specifications;
storing the formal specifications and the test sources in a repository;
compiling the test sources into an implementation language used by the PIUT to produce tests;
executing the tests on the PIUT; and
analysing results of the tests to verify the PIUT.

31. The method as claimed in claim 30, wherein
the step of generating test sources includes the step of re-generating test sources using the stored formal specifications and test sources when the PIUT is modified; and

the step of executing the tests uses tests complied from the re-generated test sources on the modified PIUT.

32. The method as claimed in claim 30, wherein the step of executing tests comprises the step of using the generated test suite when only implementation of PIUT is modified.

33. The method as claimed in claim 30, wherein
the step of executing the tests includes the step of generating and collecting trace information of the tests; and
the step of analysing results comprises the step of estimating completeness of test coverage using the trace information.

34. The method as claimed in claim 30, wherein
the step of generating test sources comprises the steps of:
generating test driver sources which are used to produce test drivers that execute tests and analyse results of the tests in implementation environments of the PIUT;
generating test case parameter sources which are used to produce test case parameters used by the test drivers for test execution; and
creating test plans for carrying out tests based on the formal specifications; and
the step of compiling comprises the steps of:
compiling the test driver sources into the test drivers; and
compiling the test case parameter sources into the test case parameters.

35. The method as claimed in claim 34, wherein the step of executing the tests includes the steps of:
initializing the PIUT;

calling the test drivers with test case parameters in accordance with the test plans; and
receiving results of the tests.

36. The method as claimed in claim 34, wherein the step of generating test driver sources comprises the steps of:

generating basic driver sources by analysing the formal specifications for each PIUT; and

generating script driver sources by creating skeletons describing general scheme of the script driver and generating script driver sources based on the formal specifications and the skeletons; and

the step of compiling the test driver sources comprises the steps of:

compiling the basic driver sources into basic drivers; and

compiling the script driver sources into a script driver.

37. The method as claimed in claim 36, wherein the step of executing the tests comprises the steps of:
executing the test plans;
launching the script driver by one of the test plans;
calling one of the basic drivers by the script driver; and
calling a PIUT by the basic driver which executes the test on the PIUT.

38. The method as claimed in claim 30, wherein the step of executing the tests further comprises the step of updating the repository.

**ABSTRACT**

A verification system for a procedure interface is generated by using formal specifications of the procedure interface and generating test suites from the formal specifications. The formal specifications are independent from implementation of the procedure interface. Accordingly, the test suites are also implementation independent. Thus, a verification system comprising the test suites may be efficiently produced for verifying the procedure interface of different implementations.

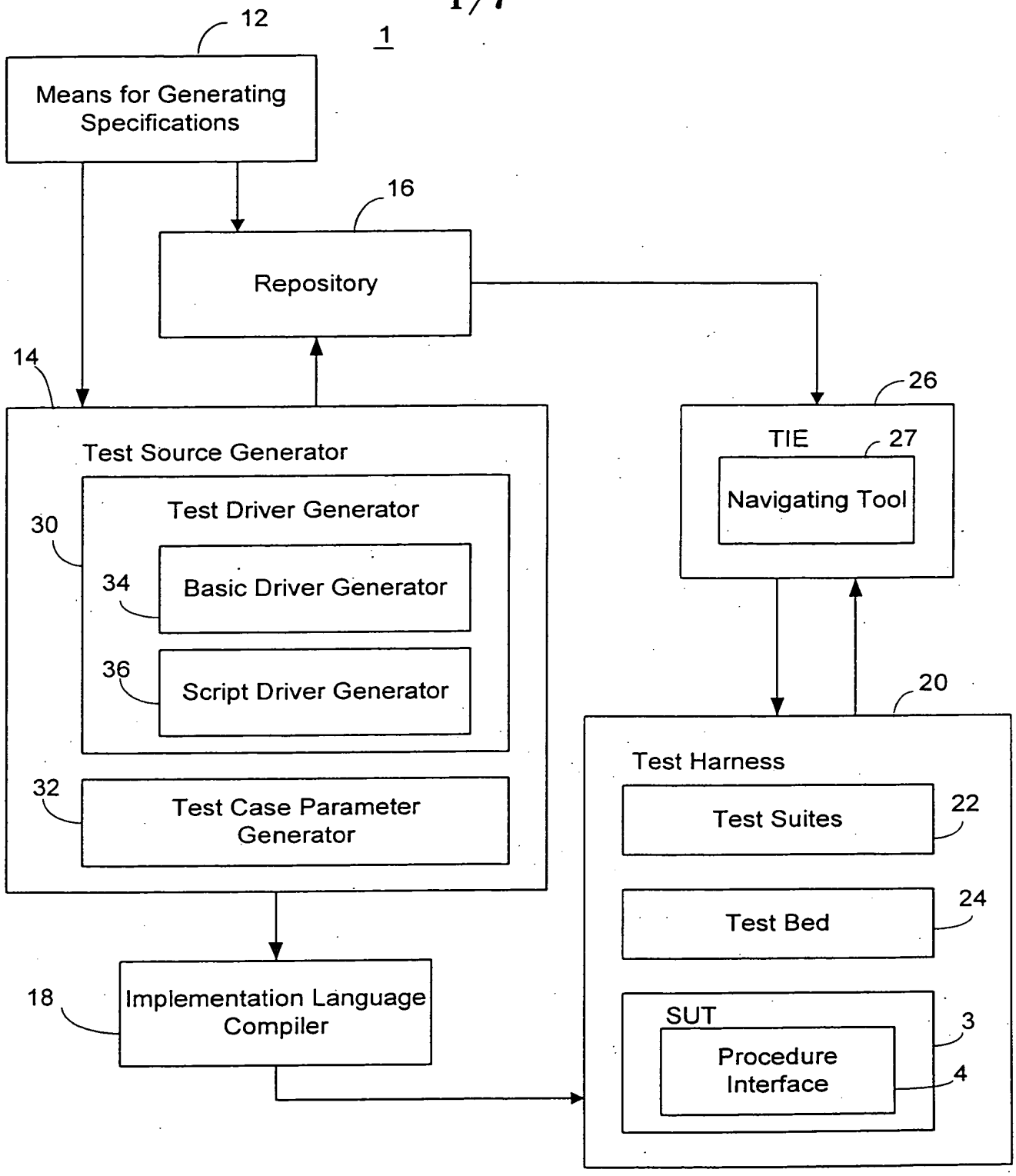
1
12
Means for Generating Specifications
16
Repository
14
Test Source Generator
Test Driver Generator
30
34
Basic Driver Generator
36
Script Driver Generator
32
Test Case Parameter Generator
18
Implementation Language Compiler
26
TIE
27
Navigating Tool
20
Test Harness
Test Suites
22
Test Bed
24
SUT
3
Procedure Interface
4
2

FIGURE 1

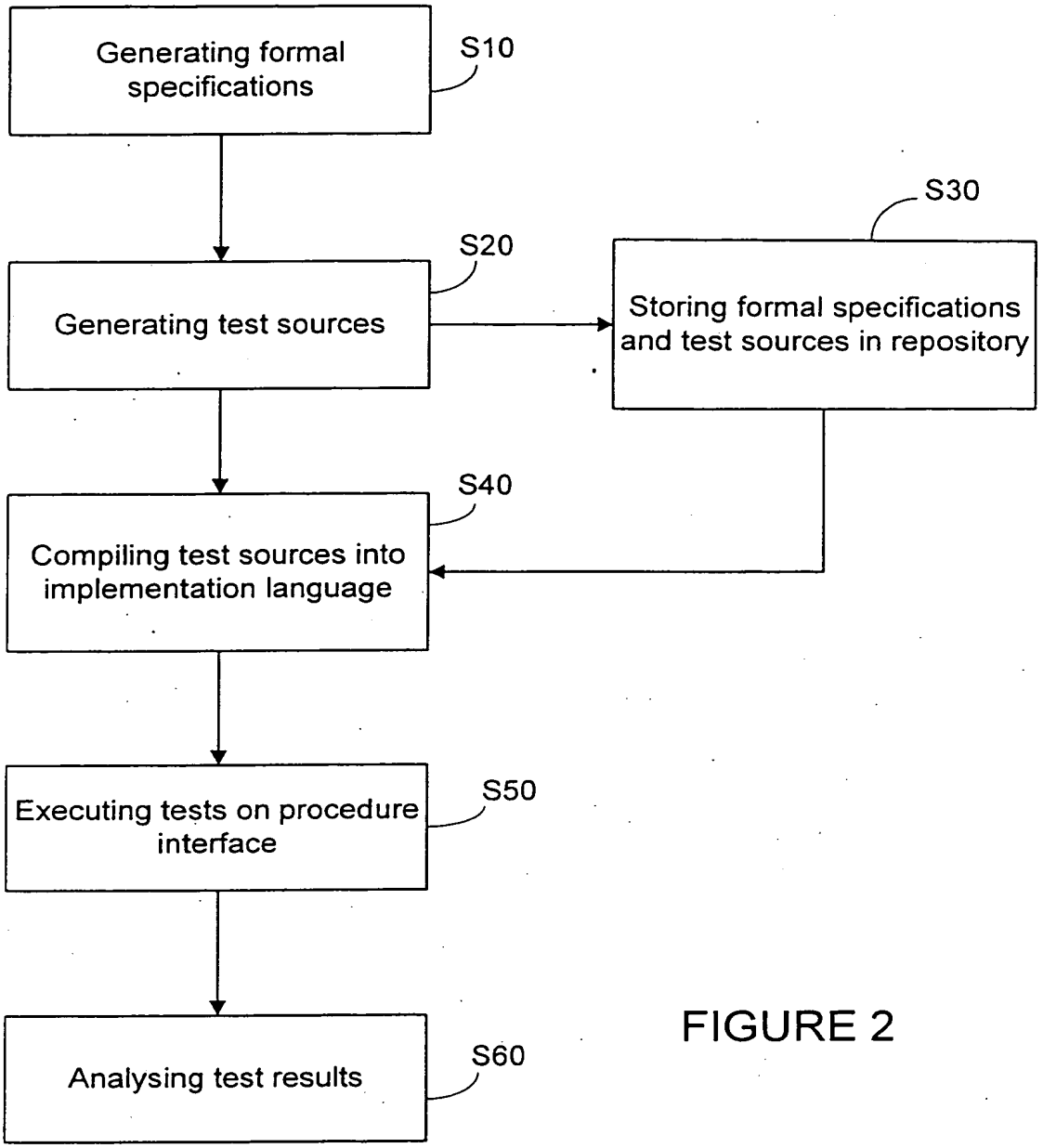
Generating formal specifications
S10
Generating test sources
S20
S30
Storing formal specifications and test sources in repository
Compiling test sources into implementation language
S40
Executing tests on procedure interface
S50
Analysing test results
S60

FIGURE 2

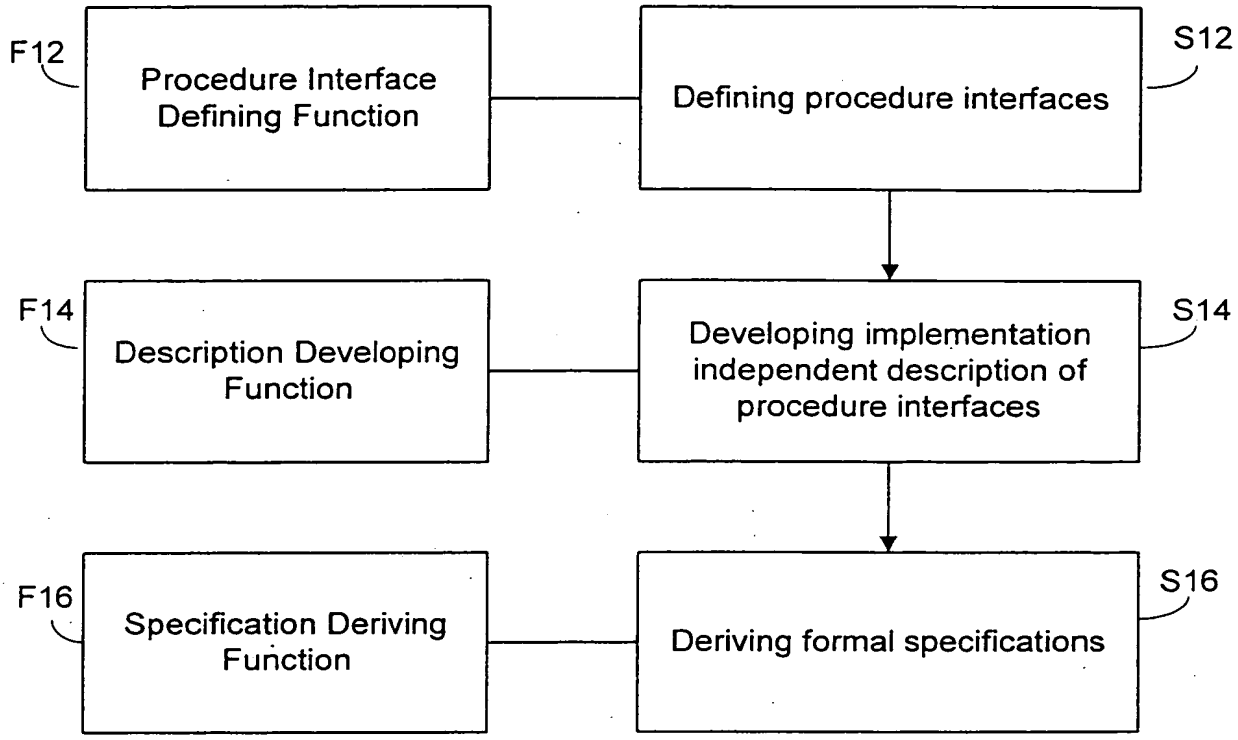


FIGURE 3

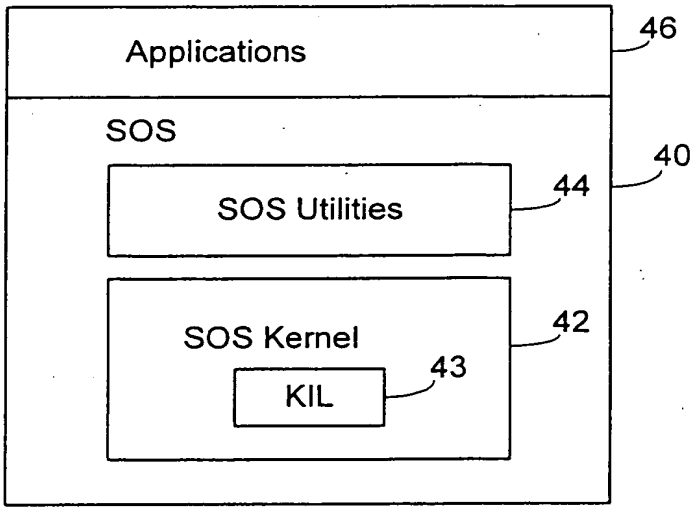


FIGURE 4

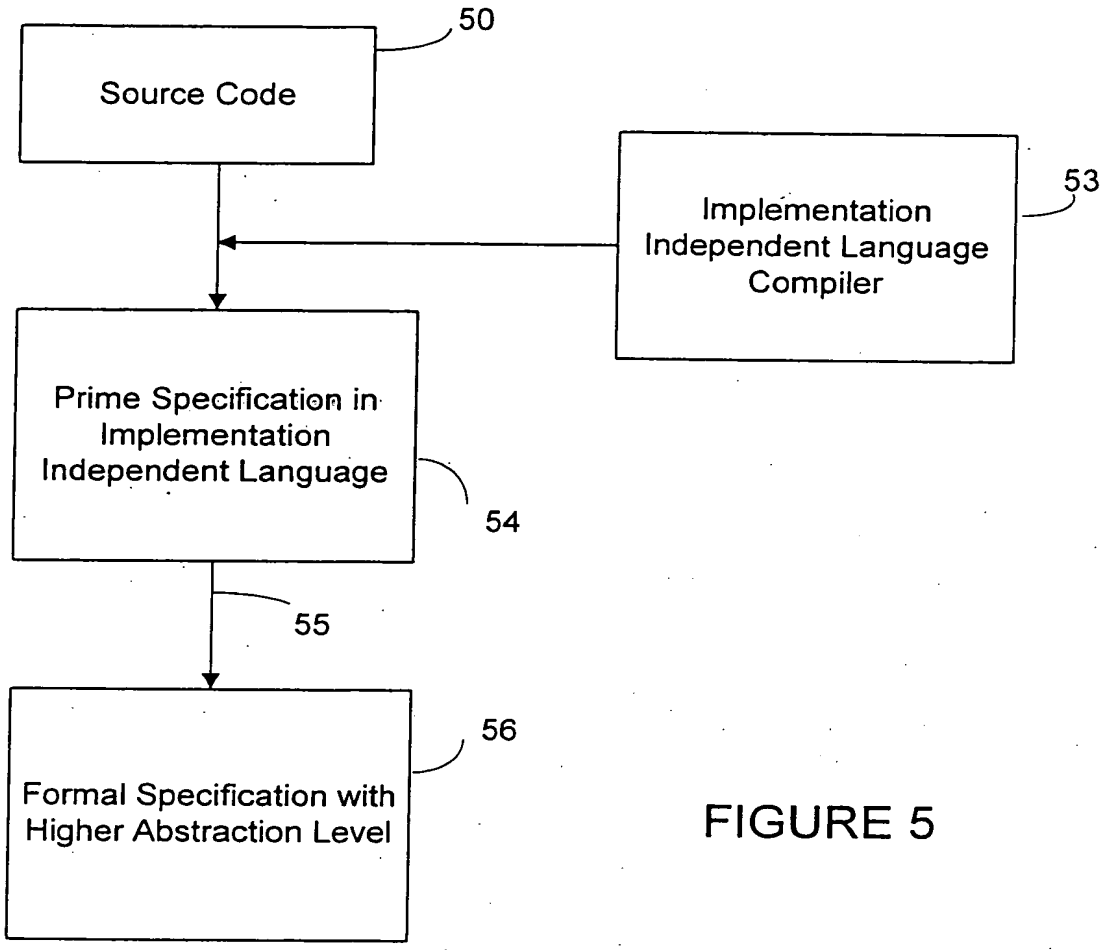
Source Code
50
Implementation Independent Language Compiler
53
Prime Specification in Implementation Independent Language
54
55
Formal Specification with Higher Abstraction Level
56

FIGURE 5

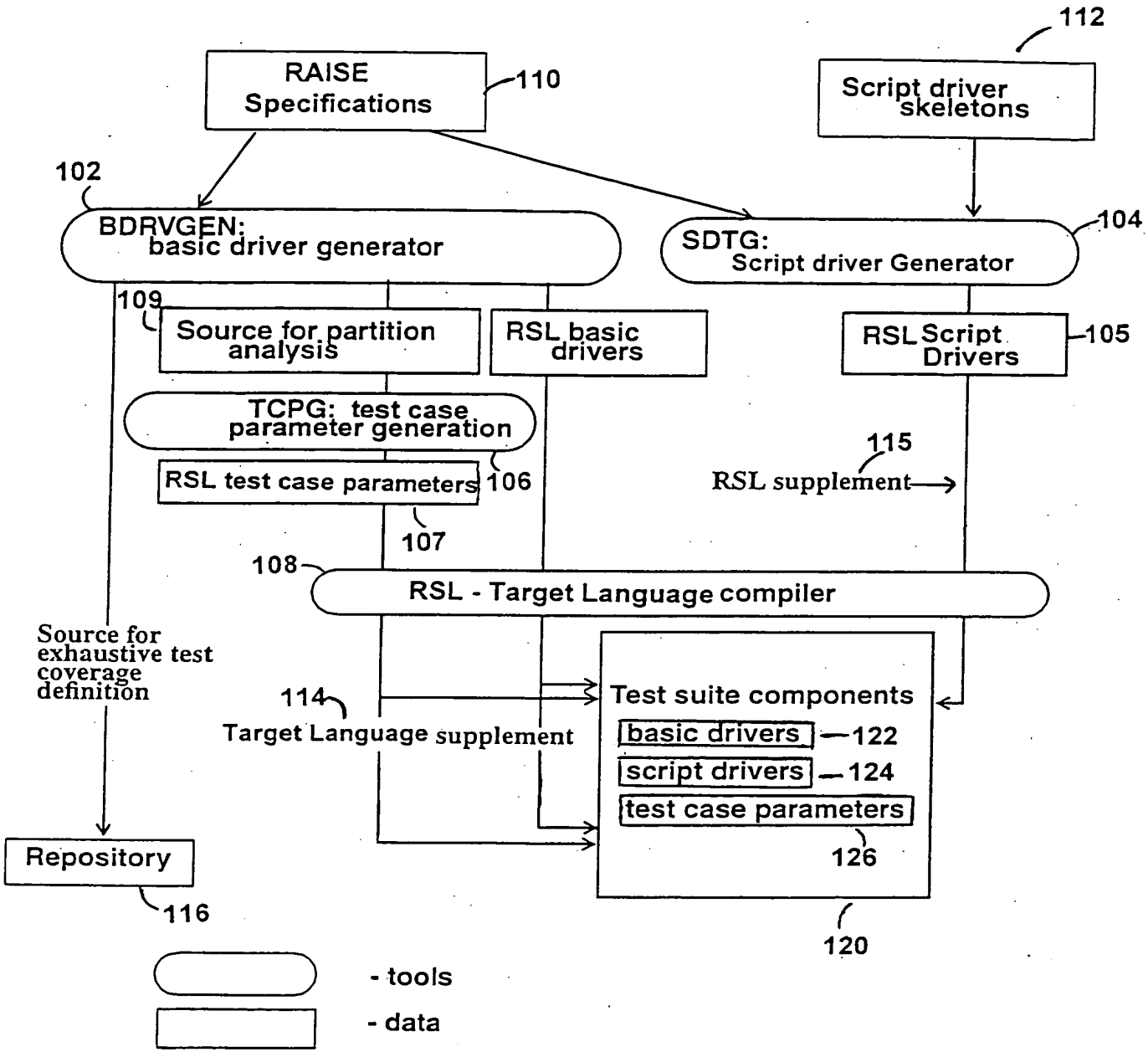
RAISE Specifications
110
Script driver skeletons
112
102
BDRVGEN: basic driver generator
SDTG: Script driver Generator
104
109
Source for partition analysis
RSL basic drivers
RSL Script Drivers
105
TCPG: test case parameter generation
RSL test case parameters
106
115
RSL supplement
107
108
RSL - Target Language compiler
Source for exhaustive test coverage definition
114
Target Language supplement
Test suite components
basic drivers
122
script drivers
124
test case parameters
126
Repository
116
120
- tools
- data

FIGURE 6

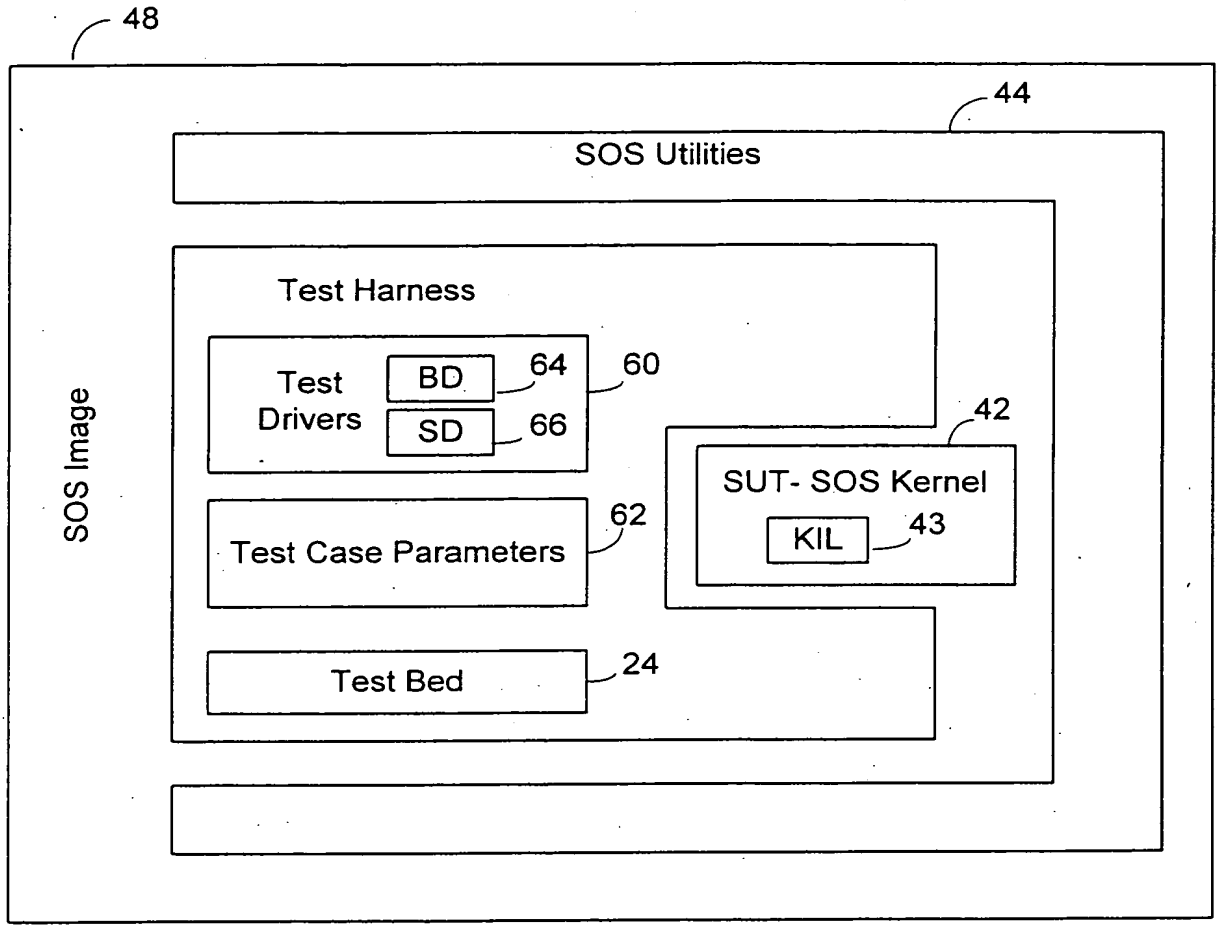
48
SOS Image
44
SOS Utilities
Test Harness
Test Drivers
BD
64
SD
66
60
Test Case Parameters
62
42
SUT- SOS Kernel
KIL
43
Test Bed
24

FIGURE 7

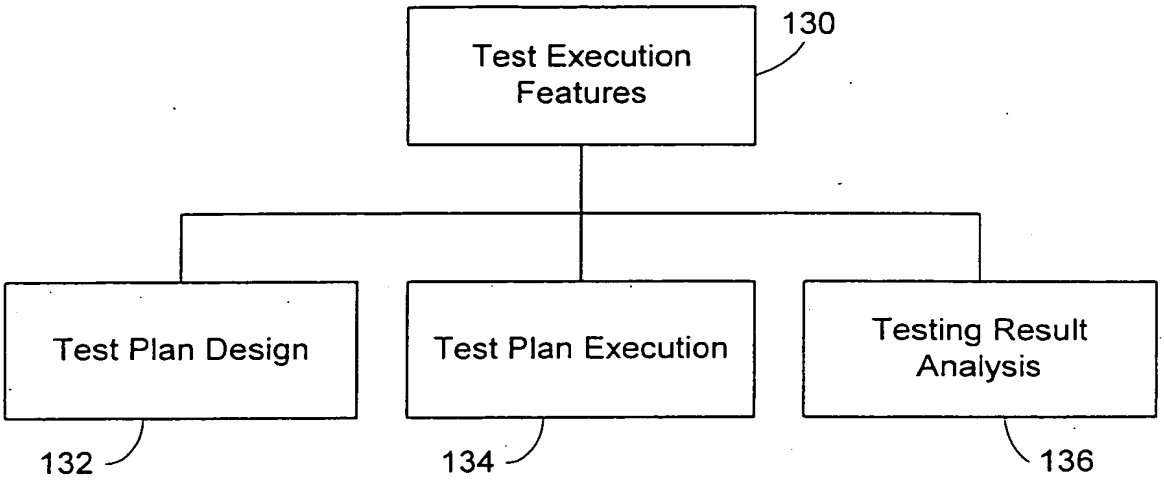


FIGURE 8

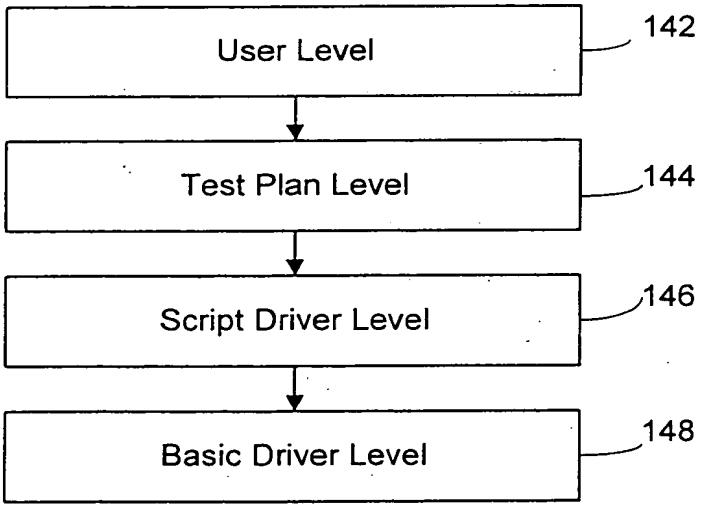


FIGURE 9